An evaluation of standard retrieval algorithms and a weightless neural approach

Conference Paper · February 2000

DOI: 10.1109/IJCNN.2000.861533 · Source: IEEE Xplore

2 authors, including:

Victoria Hodge
The University of York
59 PUBLICATIONS **2,223** CITATIONS

SEE PROFILE

An Evaluation of Standard Retrieval Algorithms and a Weightless Neural Approach

Victoria J. Hodge
Dept. of Computer Science,
University of York, UK
vicky@cs.york.ac.uk

Jim Austin
Dept. of Computer Science,
University of York, UK
austin@cs.york.ac.uk

Abstract

Many computational processes require efficient algorithms, those that both store and retrieve data efficiently and rapidly. In this paper we evaluate a selection of data structures for storage efficiency, retrieval speed and partial matching capabilities using a large information retrieval dataset. We evaluate standard data structures, for example inverted file lists and hash tables but also a novel binary neural network that incorporates superimposed coding, associative matching and row-based retrieval. We identify the strengths and weaknesses of the approaches. The novel neural network approach is superior with respect to training speed and partial match retrieval time.

Many computational implementations require algorithms that are storage efficient, may be rapidly trained with data and allow fast retrieval of selected data. Information retrieval requires the storage of massive sets of word to document associations to allow the documents matching query terms to be retrieved. This often requires partial matching of M of N query terms. Many methodologies have been posited for storing these associations, see for example, [Knu68]: including inverted file lists, hash tables, document vectors and superimposed coding techniques. We only compare 'perfect' techniques; those that preserve all word to document associations. We analyse an inverted file list (a version of which is used in the Google search engine [BP98]), against a hash table and against a binary associative memory [Aus95]. In all cases we evaluate the algorithms in their standard form without any sophisticated improvements to provide a valid comparison of the approaches. We evaluate the algorithms for storage use; training speed[1] and partial matching capabilities.

We utilise the Reuters 21578 Newswire text corpus for the investigations. The dataset is large with on average 100 words per document. This allows the extraction of a large set of word-to-document associations for a thorough evaluation of the algorithms. We cleaned the text corpus and extracted 9491 words from the 18249 documents left giving 1230893 word-to-document associations. We analyse the three data structures for storage efficiency, training speed and partial match retrievals. We assume that all words searched are present; we do not consider error cases in this paper, as we are evaluating speed and capacity.

1 Data structures

1.1 Inverted File List (IFL)

For the inverted file list we use an array of words, sorted alphabetically and linked to an array of lists. This data structure minimises storage and provides flexibility. The array of words provides an index into

[1] training time and thus speed is implementation dependent. To minimise variance, we preserve as much similarity between the data structures as possible particularly during training - see section 1 for details

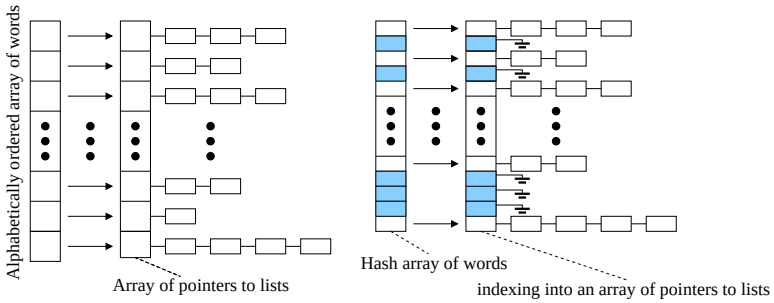

Figure 1: Diagram showing the IFL and hash data structures. We implemented two linked data substructures to preserve similarity between the data structures. The speed of training and retrieval are dependent on implementation. By maintaining similarity, we enable comparisons between the two data structures.

the array of lists (see figure 1). A word is passed to an indexing function (binary search through the alphabetically sorted array of words) that returns the position of the word and its document list in the arrays. The lists are only as long as the number of documents associated to minimise storage but yet can easily be extended for new associations. The retrieval of the index into the array of lists using binary search is $O(\log n)$ so the data structure is a trade-off between minimising storage but providing slower access, the hash table below is $\Omega(1)$ for access in comparison.

The IFL and hash table achieve partial matching through the addition of an array of documents and counters. In this paper the documents were identified by integer IDs so we exploit this to provide an index into an array of counters. The counter stored at position ID is the number of times that document ID has been retrieved. A single pass through the array, once retrieval is complete, will retrieve the documents that have matched the required number of words (M of N).

1.2 Hash Table

The hash table is used to maximise the retrieval speed. An array of words is linked to an array of pointers that point to linked lists (see figure 1) to minimise storage and maintain flexibility while a hash function generates the indices. The complexity of an average hash operation is constant time $\Omega(1)$. We use *closed hashing* where no more than one item is stored at each hash location. If this array location is empty, the new item can be inserted. However, if this location is already occupied, an alternative strategy must be used. The worst-case upper bound for the hash operation complexity is thus $O(n)$. The price for such a speed up in access time is memory usage. The hash table used in the paper degrades as the table fills and thus must be kept less than 50% occupied otherwise there are too many collisions. For efficiency reasons, we must minimise collisions by using an initial hash function that does not favour any particular set of locations. We empirically evaluated various hash functions for strings *(char * word)* and found that Horner's Rule with factor 131 (adapted from Sedgewick [Sed92]) produced the least number of collisions.

```
for (sum=0; *word; word++) {
    sum = (sum*131) + *word; }
return (sum % hashTableSize); }
```

For double hashing, $\Omega(n^2)$ probe sequences are used rather than $\Omega(n)$ for linear or quadratic probing (see [CLR90]) improving performance as more possible locations are examined. We selected a prime number size (20023) to ensure that the modulo function (%) is not followed by an integer with small divisors else the hash algorithm will be pre-disposed to clustering [Amm98]. Also, if $hashTableSize$ and

$hashTableSize - 2$ are not relative primes at the least, they will have a greatest common divisor (gcd) and only $\frac{1}{gcd}$ of the table will be probed (see [CLR90]) for vacancies both 20023 and 20021 are prime.

Again for partial match we employ the additional document data structure used for the IFL.

1.3 Advanced Uncertain Reasoning Architecture (AURA)

AURA [Aus95] is a collection of techniques that may be implemented in a modular fashion and utilises Correlation Matrix Memories (CMMs, [Aus95]) to map inputs to outputs (see fig **??**) through a learning rule, similar to a hash function. AURA does not suffer from the lengthy training problem of other neural networks; training is a one shot (single epoch) process and AURA is able to match only partial inputs. Storage is efficient as patterns are stored distributively and new inputs do not require additional memory allocation, they are overlaid with existing trained patterns [TA]. The AURA modular neural network uses binary weights and inputs. Learning is supervised as the outputs are known and guide the learning process. The words form the inputs and the documents the class patterns (outputs). The words, are initially translated to binary patterns by the lexical token converter (the neural network requires binary data inputs). The binary pattern for each token has a preset constant number of bits set to 1 in a fixed-length binary array. The documents form the class patterns, again the document ID is translated by the lexical token converter to a binary bit vector with a predefined number of bits set. We use orthogonal vectors with 1 bit set to prevent false positive matches during retrieval (see [Knu68]).

1.3.1 Training the network

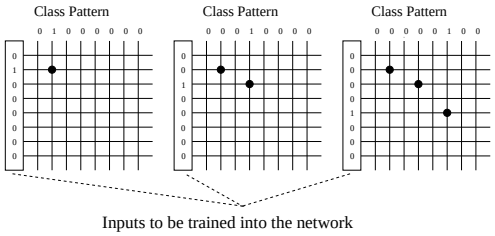

Figure 2: Diagram showing three stages of network training

The diagram (figure 2) shows a CMM after 1,2 and 3 patterns have been trained. The CMM is set to one where an input and class pattern (row and column respectively) are both set (see figure 2). The training process is $\Omega(n)$ as there is one association trained into the CMM per word-document pair.

1.3.2 Recalling from the network

This process is essentially similar to the training process except only the word pattern is input. The columns are summed and the activation of the network thresholded using the Willshaw threshold (see [Aus95]) to produce an output vector (see figure 3). The vector retrieved represents the matching document for the input word presented to the network during recall.

If we wish to retrieve multiple word matches, rather than serially matching the bit vectors, AURA replicates parallel matching. The bit vectors matching the required word inputs are superimposed, forming a single vector (see figure 3). Thus multiple word matching is $\Omega(1)$ with respect to input epochs. The previous two data structures are $\Omega(n)$. For partial match, AURA exploits the fact that all words in the IR system have a definite mapping from the words to the documents. If only M of N words ($M < N$)

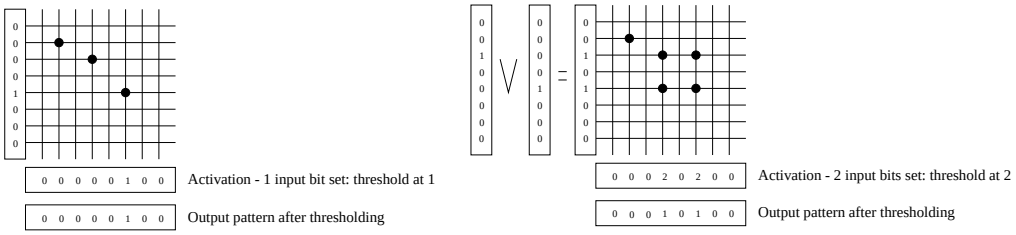

Figure 3: Diagram showing system recall for single and superimposed inputs. The single input pattern has 1 bit set so the CMM is thresholded at 1. The superimposed input has 2 bits set so the Willshaw threshold is set to 2.

in the input must match then the Willshaw threshold is set at $M \cdot B$ where B is the number of bits set in each input vector. The outputs from multiple word retrievals will be superimposed in the output vector.

The outputs for single and multiple word matches must be identified. A list of valid outputs is held in a content-addressable memory and matched in the lexical token converter - binary to data. The time for this process is proportional to the number of bits set in the output vector Θ(bits set), there will be one matching word per bit set for orthogonal (single bit set) vectors.

1.4 Analyses

All analyses were performed on a SGI Origin 2000 with the following specifications:
32 X 180 MHZ IP27 Processors (MIPS R10000).
Main memory size: 8192 Mbytes.
All data structures were compiled with the CC compiler using -Ofast (fast binaries) and -64 (64 bit). The algorithms were run with the command *runon x <algorithm>* and one process from each data structure was run in parallel to minimise variance.

The following analyses were performed on the data structures:

1. The memory usage for each algorithm was calculated using the C/C++ sizeof() utility.

2. The training times were calculated using the C/C++ clock() function. The training time was the time to read in the list of words, input them to the data structure and read in each of the word-to-document associations adding the appropriate links in the data structures (document IDs are added to the front of the lists in the IFL and hash table for speed). For the CMM the list of documents was also read into the data-to-binary converter prior to adding the associations.

3. Partial Match. Each data structure was trained a priori. The list of the most frequent words; those that occur in at least 20% of the documents was read and the matching documents for each word retrieved in turn. For the IFL and hash table the appropriate document counter was incremented as each document was returned. After all words had been matched, the document list was traversed to find all documents matching $> M$ words. For the CMM, the binary patterns for the words were retrieved, superimposed and input. The output was thresholded at the appropriate value and the matching documents retrieved from the binary-to-data lexical token converter. The partial match retrieves the documents that match more than 10, 11, ... and more than 37 of the most frequently occurring words to test partial matching. There are 38 words in the word set and the 18249 documents match between 0 and 38 words. For all data structures the output to the file is identical.

1.5 Results

1.5.1 Memory Usage

	IFL	Hash Table	CMM
Total Memory Usage in bytes	10945622	11556478	14200718

1.5.2 Training Times

	IFL	Hash Table	CMM
Average Training time for 20 training epochs in secs	1965.365	1570.48	98.31

1.5.3 Partial Match

Figure 4 shows a graph of the speedup of the average retrieval times (averaged over 20 retrievals) for each data structure. The data structure were timed for retrieving the documents that match more than 10 to more than 37 of the most frequently occurring words in the text corpus. The graph shows the time for each data structure / the time for the CMM.

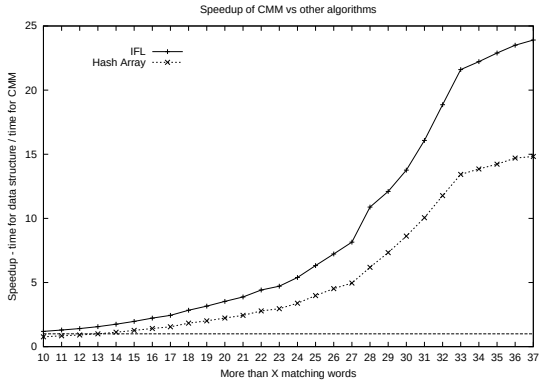

Figure 4: Graph of the speedup of the CMM versus the other data structures when retrieving highly saturated words. Speedup = time for data structure / time for CMM retrieval.

2 Analysis

2.1 Memory Usage

The memory totals for the 3 algorithms are in ascending order: IFL, hash table and CMM. As stated previously, the inverted file list will use the least memory but, as can be seen from the subsequent analyses, is slower. The CMM memory usage is 1.3 times higher than the inverted file list. However, none of the memory usage statistics is significantly larger than the others.

2.2 Training Times

There is a significant difference between the training times. The IFL is the slowest to train as expected. Each word search requires a binary search through the entire array $O(\log n)$ before the document may be appended to the word's list. The training times for the hash table is quicker as we would expect. Training finds the hash location for each word ($\Omega(1)$) and appends the document to the corresponding document

list. By far the quickest to train is the CMM. The CMM does not require the lengthy word search. Even the hash function $\rightarrow O(n)$ if there are collisions. The CMM simply associates and recalls the word and document in one step.

2.3 Partial Match

The IFL is slower than the hash tables in all instances of partial matching due to the binary search $O(\log\ n)$ through the array to locate the words prior to finding the associated documents. The hash table is faster than the IFL in all instances. The CMM is slower than the hash table for low frequency partial match but for higher frequency partial match the CMM excels. The time curve for the CMM starts above the hash table but falls below for 'more than 13'. The time difference between the CMM and the has table will level off and eventually fall as the y-axis forms a lower bound asymptote to the CMM curve and the retrieval speed of the hash table will continue to fall thus slowly approaching the CMM curve. However, the hash table will only approach the CMM slowly and may never reach it so the CMM is preferable for partial matching.

3 Conclusion

For partial matching the CMM performs best. Although the memory usage is higher (but not significantly so) the retrieval speed is superior; the greater the number of words to be matched at each retrieval, the more superior the CMM is. We recommend the CMM for word to document association storage and retrieval as the approach is superior with respect to training time and retrieval time with only a slightly higher memory usage. The CMM enables single epoch training, requiring only one pass through the list of word-document associations unlike other neural networks that require multiple epochs to enable the network to settle. The superimposition of the input vectors allows one-shot multiple word retrieval and partial match - emulating parallel retrieval. We also note that more word-document associations could be added to the existing CMM without significantly increasing the memory usage due to the superimposed storage. For the other data structures evaluated, additional associations would increase the memory use of the list array with one additional list node for each additional document association.

References

[Amm98] L. Ammeraal. *Algorithms and Data Structures in C++.* John Wiley & Sons, Chichester, England, 1998.

[Aus95] Jim Austin. Distributed associative memories for high speed symbolic reasoning. In R. Sun and F. Alexandre, editors, *IJCAI '95 Working Notes of Workshop on Connectionist-Symbolic Integration: From Unified to Hybrid Approaches*, pages 87–93, Montreal, Quebec, August 1995.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *7th International World Wide Web Conference*, 1998.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 3. 1968.

[Sed92] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[TA] Mick Turner and Jim Austin. Matching Performance of Binary Correlation Matrix Memories.